A first year progress report on:

Development of a Dynamically Configurable, Object-Oriented Framework for Distributed, Multi-modal Computational Aerospace Systems Simulation

By

Abdollah A. Afjeh, Ph.D.
John A. Reed, Ph.D.

Department of Mechanical, Industrial and Manufacturing Engineering
The University of Toledo

October 30, 2000

Summary

This report describes the progress made in the first year (Sept. 1, 1999 to Aug. 31, 2000) of work at The University of Toledo under the NASA Information Technology (IT) Program grant number NAG-1-2244. This research is aimed at developing a new and advanced simulation framework that will significantly improve the overall efficiency of aerospace systems design and development. This objective will be accomplished through an innovative integration of object-oriented and Web-based technologies with both new and proven simulation methodologies. The basic approach involves three major areas of research:

- Aerospace system and component representation using a hierarchical object-oriented component model which enables the use of multimodels and enforces component interoperability.
- Collaborative software environment that streamlines the process of developing, sharing and integrating aerospace design and analysis models.
- Development of a distributed infrastructure which enables Web-based exchange of models to simplify the collaborative design process, and to support computationally intensive aerospace design and analysis processes.

Research for the first year dealt with the design of the basic architecture and supporting infrastructure, an initial implementation of that design, and a demonstration of its application to an example aircraft engine system simulation.

Year 1 Accomplishments

Work was begun in several areas during the first year of this three year grant. Major results are summarized below. A more comprehensive description of the methodology and initial accomplishments, along with an overall vision statement of our long term research goals, was published in Ref. 1.

Common Model Framework

An object-oriented domain framework for representing aerospace components. systems and subsystems has been developed. The framework, which we call the Common Model Framework (CMF), provides the foundation for the Denali[1] aerospace simulation system. The framework formalizes an approach for abstracting aerospace domain physical structure and mapping it to the computational domain. As shown in Figure 1, aerospace systems, such as an aircraft, are hierarchically decomposed (Fig. 1b) into subsystems and components (e.g., fuselage, engines, vertical stabilizer, etc.), which are then abstracted using a control volume approach (Fig. 1c). The control volumes provide both a physical geometry representation as well as a convenient mechanism for mathematical modeling. Each component can be further decomposed to identify more basic components. The most basic components may be represented in the computational domain by an object class. Following the Denali CMF architecture, the more basic classes can be instantiated and the various objects combined to form more complex objects. This object composition provides a powerful and flexible mechanism for modeling and simulating aerospace systems, allowing complex aerospace systems to be composed in the same familiar manner as the physical system.

There are four basic entities in the Denali architecture: *Element, Port, Connector* and *DomainModel* (see Fig. 1d). The Java™ interface **Element** represents a control volume, and defines the key behavior for all engineering component classes incorporated into Denali. It declares the core methods needed to initialize, run and stop model execution, as well as methods for managing attached **Port** objects. Classes implementing this interface generally represent physical components, such as a compressor, turbine blade. or bearing, to name a few. However, they may also represent purely mathematical abstractions such as a cell in a finite-volume mesh used in a CFD analysis. This flexibility permits the component architecture to model a variety of physical systems.

An **Element** may have zero or more **Port** objects associated with it. The **Port** interface represent a surface on a control volume through which some entity (e.g., mass or energy) or information passes. **Ports** are generally classified by the entity being transported across the control surface. For example, a Compressor object might have two **FluidPort** objects—representing the fluid boundaries at the Compressor entrance and exit—and a **StructuralPort** object, representing the control surface on the Compressor through which mechanical energy is passed (from a driving shaft).

---

1. Not an acronym.

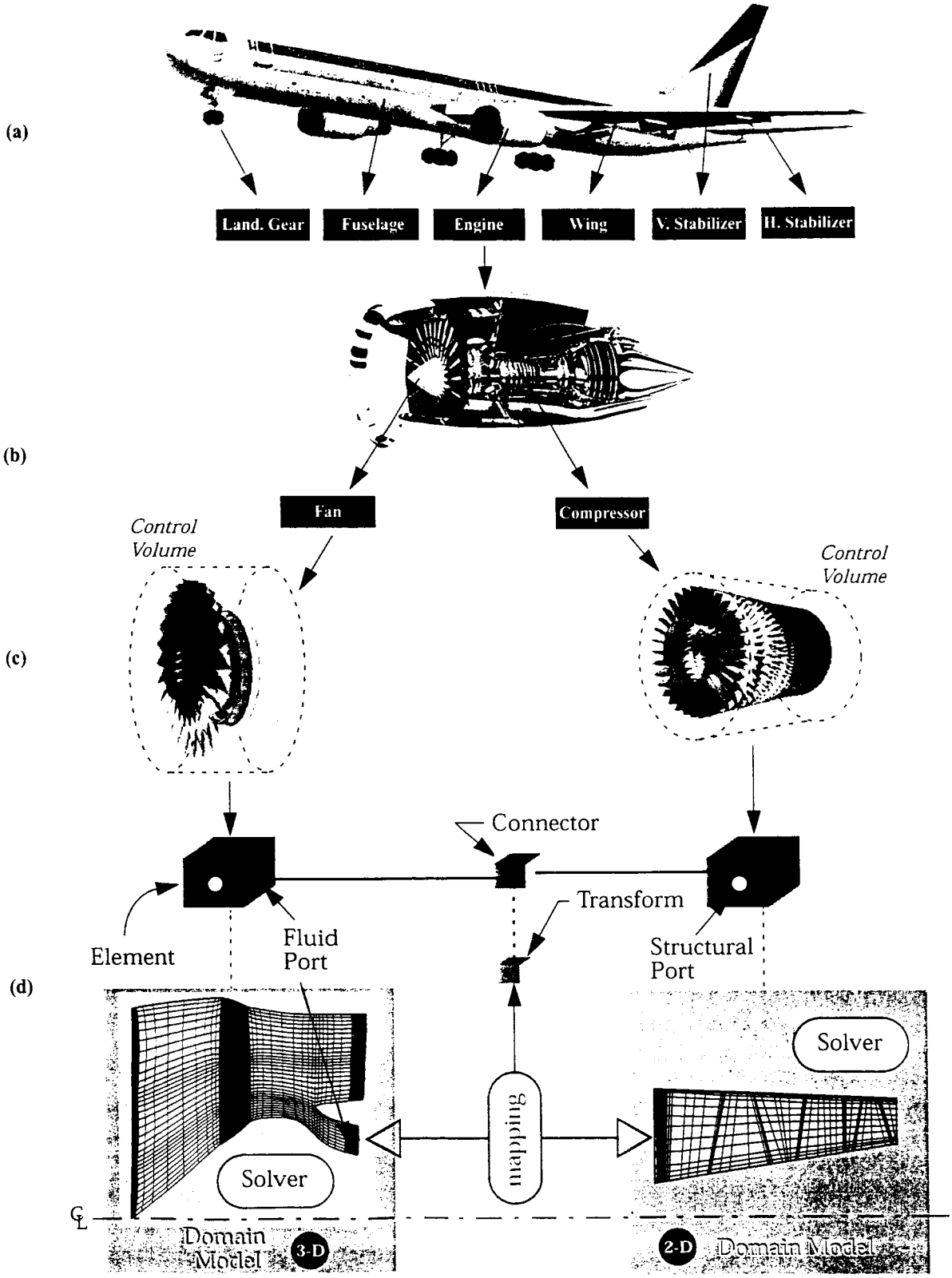

Figure 1: Mapping of aerospace physical domain to computational framework.

The common boundary between consecutive control volumes is represented by a **Connector** object. The interface **Connector** permits two **Element** objects to communicate by passing information between connected **Port** objects (see Fig. 1d). It is also responsible for data transformation and mapping in situations where the data being passed from **Ports** is of different type. The need for such data transformation can range from simple situations, such as conversion of data units, to very complex ones involving a mismatch in model fidelity (e.g., connecting a 2-D fluid model to a 3-D fluid model) or disciplinary coupling (e.g, mapping structural analysis results from a finite-element mesh to a finite-volume mesh used for aerodynamic analysis). For all but the simplest cases, the algorithms needed to perform the data transformation or mapping will tend to be very complex. To improve reusability, **Connector** delegates transformation/mapping responsibilities to a separate **Transform** object (see Fig. 1d) which encapsulates the necessary intelligence to expand/contract data and map data across disciplines.

The **DomainModel** represents the mathematical model used to define component behavior. During component design and analysis, many different models (i.e., multimodels) are used. During preliminary design the models are relatively simple and may be solved analytically or using basic numerical methods. However, models used in latter phases of design can be quite complicated. In these cases, approximate solutions are obtained by discretization of the equations on a geometrical mesh and applying highly specialized numerical solvers. The presence of these complex mathematical models and the numerical tools needed to solve them suggest that it is desirable to encapsulate these features and remove them from the **Element** structure. This enhances the modularity of **Element**, allowing new **Element** classes to be added without regard to the mathematical model used, and conversely to add new models without affecting the **Element** class. To achieve this, Denali utilizes the Strategy design pattern to encapsulate the mathematical model in a separate object. The benefit of this pattern is that families of similar algorithms become interchangeable, allowing the algorithm—in this case the **DomainModel**—to vary independently from the **Elements** that use it. This admits the possibility of run-time selection of an appropriate **DomainModel** for a given **Element**; however, this is currently not used in Denali. Furthermore, encapsulating the **DomainModel** in a separate object also encourages the "wrapping" of pre-existing, external software packages. For example, the Fan **DomainModel** in Fig. 1d might "wrap" a pre-existing three-dimensional Navier-Stokes or Euler flow solver to provide steady-state aerodynamic analysis of fluid flow within the Fan. This approach allows proven functionality of existing software analysis packages to be easily integrated within an **Element**.

The standard object interfaces of the Denali CMF ensure that each component object interoperates with other component objects. This is essential for providing a stable modeling environment which allows complex models to be developed using object composition and class inheritance. Furthermore, the standard interfaces of the CMF architecture provide a "pluggable" architecture wherein new components can be added at runtime.

As an example application of the CMF, a model of a the NASA/GE Energy Efficient Engine (EEE) gas turbine aircraft engine was created. **Elements** representing the inlet, fan, compressor, combustor, shafts, turbines, nozzle and ducts in a turbofan engine were developed. The **DomainModel** for each **Element** was developed using a zero-dimensional mathematical treatment. Furthermore, only an aerothermodynamic disciplinary analysis was used. At this level of fidelity and discipline, component behavior was defined by the unsteady, space-averaged forms of the aerothermodynamic conservation equations. Empirical data, in the form of performance maps, were used to define operating behavior for rotating components, such as Compressors and Turbines. The component objects were combined using appropriate zero-dimensional fluid and mechanical **Port** and **Connector** objects. A Newton-Raphson numerical execution scheme (also provided as part of the Denali system) was used to sole the model equations and simulate both steady and unsteady engine operation. Results of the tests were validated against other existing FORTRAN gas turbine engine simulation programs.

Connection Services Framework

Aerospace design and analysis requires the interaction of many people at different geographic locations. Even if these individuals are part of the same company, today's increasingly international business environment and corporate structures requires us to assume that the participants may not be at the same location. Moreover, strategic partnerships between companies (even those competing in the same business domain) are becoming more common place requiring additional interaction across company boundaries. As a result, it is important that our simulation framework enable users to collaborate by sharing models and data in a heterogeneous work environment.

Denali supports the exchange of models through the use of *mobile code*. Mobile code is defined as program code which can be transferred from one computer to another and executed (without recompilation) on the receiving computer. An example of this is the Java byte-code which is executed on the receiving machine by a Java Virtual Machine interpreter. Denali utilizes this feature to allow designers to create, compile, verify and share Java-based component models. Following the design guidelines specified by the CMF, aerospace components are created, placed on a Web-server and downloaded to a Denali client. Once loaded to the client, the model can be combined without additional programming effort to form a new model.

In aerospace design and analysis, as in many other engineering domains, access to distributed resources is critical. The computationally intensive nature of higher fidelity analysis codes (such as Computational Fluid Dynamics) require access to high performance supercomputers or networks of workstations. Furthermore, the use of legacy code in aerospace design and analysis often require access to codes that are constrained to run on specific architectures or operating systems. As a result, it is important that our simulation framework enable users to access the appropriate computing resources for the target application.

The Denali Connection Services Framework (CSF) provides the necessary infrastructure to enable *transparent* access to distributed resources using both Web-based exchange of models, and distributed object service. Web-based models—models written entirely in Java—are created, compiled, verified, tested and placed on an HTTP web server where they can be accessed from a Denali client. Non-Java models, such as legacy FORTRAN software, which are fixed to a particular location due to code size, computing architecture or proprietary reasons, are placed on remote machines and wrapped by a Java object. This wrapper defines an interface to the legacy code and acts as a proxy, enabling the legacy code to be viewed as a local object. As with the Web-based models, the Java wrapper for the remote legacy code is placed on a Web server so that it may be downloaded to the Denali client.

The Denali client, positioned on a user's workstation or personal computer, locates available Web-based and remote models by querying one or more well-known naming or directory service. Using a Component Browser, a user can browse the objects and data stored in a naming or directory service (see bottom-right corner of Fig. 2). Denali currently supports access to common naming and directory services, such as NDS, LDAP, CORBA Naming Service (COS Naming), and RMI Registry, through the Java Naming and Directory Interface (JNDI). JNDI is an API that provides an abstraction that represents elements common to the most widely available naming and directory services. JNDI also allows different services to be linked to together to form a single logical namespace called a federated naming service. Using the Component Browser, Denali users are able to navigate across multiple naming and directory services to locate simulation data, objects and components.

Currently, we mainly use an LDAP (Lightweight Directory Access Protocol) service which provides both naming (objects are referred by their name) and directory (objects are stored in hierarchies) access. We utilize the OpenLDAP software, an open-source implementation of the LDAP protocol, running on a UNIX workstation in our lab. Rather than storing the model objects in the LDAP service, we chose to store only attributes of the component. This reduces the need to store and transfer large objects from the LDAP, and allows models to be located by searching for keywords corresponding to certain attributes. For example, for each model component, we define the class name, the model author, model creation and expiration date, and the URL of the model code, to name a few. When a component is selected from the LDAP, the Java byte-codes are downloaded from the Web server defined by the component's URL attribute. On the client machine, the byte-codes are dynamically loaded and used to create an instance of the model.

For security purposes, the Component Browser requires users to authenticate themselves before they can retrieve any information from a naming or directory service. Once authentication has been successfully completed, the user can browse or search (using attribute keywords) the entire namespace (subject to any authorization restrictions). Authentication and authorization capabilities are provided through JNDI and the Java Authentication and Authorization Service (JAAS) framework. These tools allow the Component Browser to remain independent from the underlying security

services, which is an important concern when working in a heterogeneous computing environment such as the Web.

Access and utilization of both Web-based and remote legacy models have been tested successfully using the Denali CSF. Component models for the EEE gas turbine engine model were placed on a Web server (mime1) located in our lab. Each component model, with the exception of the Combustor, was defined as a Web-based model (i.e., written in Java). For this test, a FORTRAN Combustor model, representing non-Java legacy codes, was written, compiled and placed on a second machine (mime2). A Java wrapper, acting as a proxy for the Combustor model, was written, compiled and placed on the Web server (mime1). Deployment of each component also included registering component attributes with the LDAP service running on a third machine (mime3). A Denali client, operating on a fourth machine (mime4), was then used to access and construct the EEE engine system model using the Denali Visual Assembly Framework, which is described below.

Visual Assembly Framework

The Visual Assembly Framework (VAF) provides a configurable, extensible graphical interface for constructing and editing Denali component and system models. Aerospace component objects, placed on Web servers and registered in the LDAP service are graphically manipulated in the VAF to create new models, or edit existing models. Icons, representing individual engine components (i.e., **Elements**), are selected from the Component Browser, dragged into a workspace window, and interconnected to form a schematic diagram (see Fig. 2). Dragging an icon from the Component Browser to the workspace window causes the selected software component to be downloaded from the Web server to the client machine. Components comprised entirely of Java classes are downloaded from a Web server to the local file system where the byte-codes are extracted from the JAR file, loaded into the Java Virtual Machine and instantiated for use in Denali. Components developed in other programming languages are not downloaded, but remain on the server. Instead, the proxy object, representing the component, is downloaded and used to connect to the remote component using the Java Remote Method Invocation (RMI) substrate.

Denali supports the creation of hierarchical component models, and an icon can represent both a single component or an assembly of components. A component with subcomponents is called a composite or structured component. Components that are not structured are called primitive components, since they are typically defined in terms of primitives such as variables and equations. Composite components are represented by a **CompositeElement** class, which is part of the **Element** hierarchy. The class structure, based on the Composite design pattern, effectively captures the part-whole hierarchical structure of the component models, and allows the uniform treatment of both individual objects and compositions of objects. Such treatment is essential for providing the object interoperability needed to perform Web-based model construction by composition.

Figure 2 shows a composite model representing an aircraft turbofan engine. The icon labeled Core is a composite of components which are displayed in the lower schematic. Each icon has one or more small boxes on its perimeter to represent its Ports. Connecting lines are drawn between the ports on different icons by dragging the mouse. A **Connector** object having the correct **Transform** object needed to connect the two ports is created automatically by Denali. Each icon has a popup menu which can be used "customize" the attributes of its **Element**, **Port** and **DomainModel** objects. When selected, a graphical **Customizer** object is displayed (see upper-right corner of Fig. 2), which can be used to view or edit the selected objects attributes. The visual assembly interface also provides tools for plotting (see the lower-left corner of Fig. 2), editing files, and browsing on-line documentation.

Using the VAF interface, the EEE component models were successfully downloaded from the Web server (mime1),and combined graphically to form an EEE engine model in the VAF. A Newton-Raphson numerical execution scheme (provided as part of the Denali system) was used to solve the system of equations and simulate both steady and unsteady engine operation. Results of the tests were validated against other existing FORTRAN gas turbine engine simulation programs.

Currently the VAF interface is implemented as a Java application rather than a Java applet. This was done for two reasons: 1) Java applications are easier to develop than applets, since they do not require explicit security controls (i.e., signing); and, 2) browser technology needed to run applets is not up-to-date. Also, a new product, called Java Web Start is now available (in beta form) which allows users to download Java *applications* which run on the desktop, in much the same manner as applets, but do not require a Web browser. We are currently experimenting with the Java Web Start to evaluate its use with Denali.

Publications Resulting from Work Supported by This Grant

[1] Reed, J. A., Follen, G. J., and Afjeh, A. A., "Improving the aircraft design process using Web-based modeling and simulation, "*ACM Transactions on Modeling and Computer Simulation*, Vol. 10, No. 1, 2000, pp. 58-83, (special issue on Web-based Modeling and Simulation).

Plans for Year 2

Common Model Framework

- The majority of work in year 2 will focus on the addition of geometry data to models. Specifically, we plan to work on providing direct access to CAD native geometry data. Our plan is to use a middleware layer being developed at MIT to allow us to access a variety of CAD packages using a common API. Access to CAD geometry will allow us to enhance our visualization capabilities.

- We plan to test integration of several database management systems with Denali. This had been slated for yr. 1, but was postponed until yr. 2 to more fully explore the use of new approaches to saving models, such as using XML.

- We also plan to obtain existing airframe models for study. These will be integrated within the Denali simulation system in year 3.

Connection Services Framework

- We will continue to improve non-mobile code services. Specifically, we are working on developing generalized specifications for wrapping legacy codes common in the aerospace domain. These include CFD and FEA tools, as well as numerical solvers and optimizers.

Visual Assembly Framework

- We will work on integration of CFD and geometry visualization. We will examine the possibility of integrating an existing visualization tool, or creating a new Java-based visualization tool to display geometry and flow data.

- We will continue to enhance and refine our VAF design to make it more intuitive and easier to use. We hope to provide a beta version of the Denali system to users at aerospace companies and NASA centers for evaluation. Feedback from these beta testers will be used to enhance the Denali VAF (and other parts of Denali).

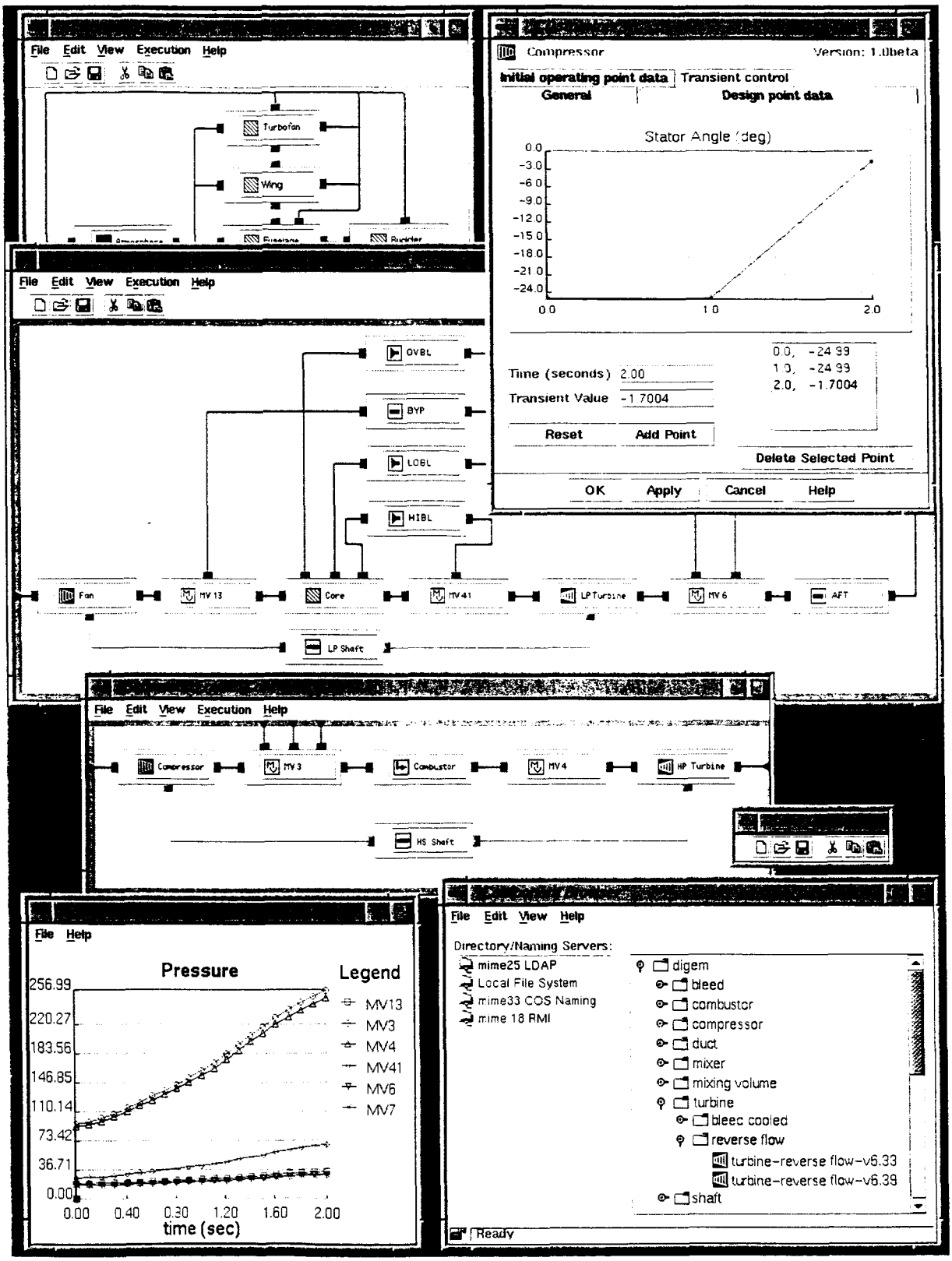

Figure 2: Denali Visual Assembly interface showing integration of engine model.